Your Turn! Part 3

Bullfrog's coding methods revolve around making life as easy as possible, so giving more time to pose in rather ridiculous positions for press photographers. Or not, as the case may be.

How to program your own games in assembler

Moving on to the third part of a continuing series which aims to teach you how to program like the pros. From the team that brought you Populous, Bullfrog coder **Scott Johnston** explores more of the details of our demo game, which is gradually building into somthing pretty playable. This month we learn how to deal with background blocks and collisons...

LAST MONTH WE had managed to get a man, the main character of our game, moving around the screen, under joystick control and properly animated. So far, so good.

But flying a little bloke around a screen wouldn't be much of a game, so it's now time to start adding some of the other elements of the game. During the course of this month we will place the platforms into the game.

The main display is usually the best part of a program to start off with, so that you can get a reasonable idea of how the finished product will look and feel. Many games start off as static background graphics and sprite animations, then the process by which the graphics become a game follows on. Which means we have started he wrong way round.

If you run the demo, your man should fall from the sky, and land at the bottom of the screen. You can't actually see any platforms, but as far as your man is concerned they are already there. Pressing fire will bounce the main sprite around, and you will find that he can stand on the platforms.

Quit out of that, and have a look inside the draw.s file. The reason we don't get all the blocks drawn on the screen is because the command

```
dbra d4,.loop_y
```

has a semi-colon in front of it. Remove this and recompile. You should now have the level

Continued overleaf

on screen. See if you can get to the top! Not that difficult, really, eh?

Fed up of that level? Then go into `move.s` and remove the semi-colon from in front of the `LEVEL` equate. Now when the program is compiled you will have a different level to try and climb. Fall down the gap at the bottom of the screen, and it will be time to quit and rerun the program.

You can change several of the equates to make the 'game' easier or harder to play. GRAVITY is the speed at which you slow down on your jump. JUMP is how high you will go. LEFT_FOOT and RIGHT_FOOT control how far off a platform your man can stand - decrease LEFT_FOOT and increase RIGHT_FOOT to make it easier. The rest you already know about.

That's great, but you may well be asking yourself 'how does it all work?' Well, the two routines that draw the block background are _draw_blocks and block_draw.

The first scans through the map data held inside data_c.s. If the number picked up is not a zero then there is a block to be drawn. This block number, as well as the x and y position of the block, is then passed to the block_draw routine. The first part of the drawing routine sets up where to draw on the screen. The loop picks up four words of data, and places them down in the right place on the screen. At present this is only a block draw, which means that anything behind it is not visible. To change it so that you could see through, you would have to create a mask for the data.

GAME DESIGN OVERVIEW

To turn this into a platform game, we need to know the order in which we have to do things. Listed below is a simple pseudo-code example of the processing order. It includes any objects that need to be picked up and any enemies that need to be drawn.

1 Initiate level.

2 Clear the screen.

3 Draw blocks.

4 Draw objects.

5 Draw bad guys.

6 Draw hero.

7 Move hero – includes x,y and animation of the sprite for our man.

8 Move bad guys – again this also includes x,y and animation.

9 Collision check on objects – has the hero touched an object?

10 Collision check on baddies – has our man just touched an enemy?

As you can see, we still have quite a lot of work to do. But saying that, now that we have broken down the task into easy sections we can bolt on new bits every month over the next couple of months.

THE SCREEN

In low-resolution 16-colour screens, each pixel is stored as four bits. Unfortunately the bits are not stored consecutively, but as a single bit in each of four words. These words are known as bit-planes.

The first word of screen memory is the first word of plane 0. The second word of memory is the second word of plane 0. Thus we fill the whole screen with plane 0 before we move on to plane 1.

Again the whole screen is filled, before we move on to plane 2, and then plane 3. Thus 32000 bytes are used in total. The first 8000 bytes are plane 0, the next 8000 bytes are plane 1 and so on through planes 2 and 3.

For a colour to be displayed on screen, the computer looks at the data in the bit-planes. Each pixel on screen has bits in the bit-planes set. These bits go to make up the colour of that pixel. For example: if all planes are set, the colour value is 1111 or 15.

Plane 0 sets the right-hand bit (bit 0). If we clear plane 0 then the colour we get is 1110 or colour 14.

Another example: plane 0 is set and all other planes are clear, so the colour is 0001.

THE MAP

The map is stored at the bottom of the file data_c.s. You can change this to your heart's content, and so make the game easier or harder to complete. If you feel up to it you can try to make the background scroll up and down with the man, as he jumps about. Experts only need apply for this bit. Don't worry if you can't do this, as the final version of our game will not have a scrolling background.

SNEAK PREVIEW TIME...

Bizarre situation at this point, I'm afraid. We're going to go a few steps further this month, but unfortunately we didn't have the time and the space to get the code on the disk.

so you're going to have to wait until next month to put his into practice, but for now give it a read through and see if you can understand what's going on. It's pretty easy.

The will be quite a lot of changes to the code – the number of files on the disk will change, making it easier to figure out what is going on. The files we are most interested in are these:

```
equates.i
structs.i
init.s
draw.s
move.s
data_c.s
```

Even without the code and from just the names, you should realise that three files you have not seen before are the `equates.i` `structs.i` and `init.s`. All the equates from `move.s` are going to be placed into the file `equates.i` because this helps in trying to track something down, or change a figure.

The major addition to the new program will be the ability to collect something – and to die! When you run the code, you will see a screen with five Golden Ankhs on it. To collect something our man must touch these.

When you try the code on next month's disk, you will find that the ankhs don't disappear. That is due to the fact that we, or should you, have not written the collision code yet.

OK. Re-run the code and take a dive from one of the top platforms to the floor – the man will die. That is it for that game so far – only one life at present.

The ankhs are in what is known as a structure. These are brillant things, in that you can define several variable names and control more than one thing with them. These names are held inside the structs.i file and are:

```
OBJ_ON          equ     0
;does the object exist
OBJ_STATUS      equ     0
;what state is it in
OBJ_TO_DRAW     equ     2
;Where to get x, y and frame in one
OBJ_X           equ     2
;where on the screen is it
OBJ_Y           equ     4
;y version of above
OBJ_FRAME               equ     6
;what is the frame being dis-
played
OBJ_SIZE        equ     8
;how big is each object
```

The numbers represent which bytes are used by this information. The numbers are all even because I am using word-sized variables due

NEW COMMANDS

`DBRA  d0,label`

The dbra command, or Decrement and Branch. This is a very useful command in that you can create loops easily and quickly. The number of times to loop is held as 1 less than the actual in d0. This is because the test that is performed is the equivalent of a BGE. So the similar code written without the dbra command is:

```
-move.w #5-1,d0 ;set up a counter
.loop_to_here
Do What Ever.sub.w      #1,d0   ;subtract from the counterbge
.loop_to_here   ;loop if d0 is not negative

BTST    #1,d0
```

Bit Test. This command checks whether bit 1 in register d0 is set to a one or not. The bits are numbered from the right-hand side, starting at 0 and moving up to 31 on the left.

THE COLLISON DETECTION ROUTINE

```
move.w          man_x,d0          ;pick up our man_x position
asr.w           #FOUR,d0          ;scale down to a screen co-ord
move.w          man_y,d1          ;pick up our man_y position
asr.w           #FOUR,d1          ;scale down to a screen co-ord
move.w          d0,d2             ;take a copy of x
add.w           #MAN_WIDTH,d2     ;and add the width of our man
move.w          d1,d3             ;take a copy of y
add.w           #MAN_HEIGHT,d3    ;and add the height of our man
lea             _objects,a0       ;point at our objects
move.w          #MAX_OBJECTS-1,d7 ;counter of how many objects
.looptst.w      OBJ_ON(a0)        ;is the first one present
beq.s    .next                    ;no so lets look at the next
;one
move.w   OBJ_X(a0),d4             ;yes it was, get the object x
move.w   OBJ_Y(a0),d5             ;and y values
cmp.w    d3,d5                    ;Is the bottom of the man
;greater than the top of the object
.bgt.s   .no_collision     ;yes then we cant collide with it
add.w    #ANKH_HEIGHT,d5   ;move to top of the object
cmp.w    d1,d5             ;is the top of the man less than the
;bottom of the object?
blt.s    .no_collision     ;yes then we cant collide
cmp.w    d2,d4             ;compare right of man with the left
;side of the object
bgt.s    .no_collision     ;if it is greater then we cant collide
add.w    #ANKH_WIDTH,d4    ;move to the right side of object
cmp.w    d0,d4             ;compare left of man with right side
;of the object
blt.s    .no_collision     ;if less than then we cant collide
move.w   #0,OBJ_ON(a0)     ;clear out the object as we touch
sub.w    #1,to_collect     ;reduce the number left to get
move.w   total_levels,d6   ;as compute the score
mulu     #SCORE,d6         ;that is received by getting object
add.w    d6,score          ;score = level * multiplier
move.w   #DELAY,delay      ;place a delay to stop instant
jump.next                  ;we want to get the next object
.no_collision
;because the last did not exist or we didnt touch it
.lea     OBJ_SIZE(a0),a0   ;so move the pointer to the next
dbra     d7,.loop          ;repeat until there are no more
;objects
```

to the fact that we have loads of memory to play with.

If this was a full-sized game I would probably be using byte-sized variables for anything that I could, just to save space. One thing to note, though, is that byte-sized variables tend to be slower, as they must be extended into words before they can be used. In Powermonger, for example, the people took up 50 bytes each, and there were up to 512 people to each map.

The variable name OBJ_TO_DRAW is there for the purpose of clearer movems. It is always a good idea to stick in extra labels for this kind of thing because when you try and read your code several months or even weeks later, you may not see the relevance of `movem.w OBJ_X(a0),d0/d1/d2` whereas `movem.w OBJ_TO_DRAW(a0),d0/d1/d2` is a lot easier to understand.

Adding helpful equates and labels does as much for the readability of a program as decent comments. It's a good idea to have virtually NO numeric constants in your code. All right, so there are loads of them in the demo – but that's because I was being lazy.

The memory is reserved for the objects at the end of data_c.s and again shows good use of equates. OBJ_SIZE is the number of bytes used per object, and MAX_OBJECTS is the number of objects we can have. Therefore if we change our mind on how the objects are held, or how many we can have, all we need to do is change the equates, rather than having to go back into data_c.s.

The objects are defined inside init.s which uses a jump table to decide which level we are currently on. The most obvious way to do this would have been…

```
cmp.w    #LEVEL_1,d0
beq      level_1
cmp.w    #LEVEL_2,d0
beq      level_2
cmp.w    #LEVEL_3,d0
beq      level_3
```

and so on. The problem with this is that if we had loads of levels then we would need loads of compares. Whereas a jump table does exactly the same thing faster, in less space and is just as readable. Compare that with this…

```
move.w
.jump_table(pc,d0.w),d0jmp
.jump_table(pc,d0.w).jump_tabledc
.w level_1-.jump_table
;1st leveldc.w  l e v e l _ 2 -
.jump_table
;2nd leveldc.w  l e v e l _ 3 -
.jump_table
;3rd level
```

OK, the next step is to put the collision in. In the separate panel, there's a listing of a routine should do just that, which you will need to type in. You don't have to type the comments in if you don't want to, but they will help a lot if you ever come back to the routine. The extra code goes in the bottom of file move.s after the label _collect_collision.

Typed that? Good. Sorry there is so much of the stuff. Did you understand it?

Recompile and run the program: you should now be able to pickup the Ankhs at the top of the screen. If you put a semi-colon in front of the move.w #DELAY,delay you will see (next month, I'm afraid, when you run this lot) that if you complete the level, you will instantly flash to the next one rather than waiting for a second or so.

Take a look at the panel on masking next and try that out. Then there's a few other things you can try to do for yourself.

One: change the score equate, to have massive high scores and so on. The score, incidentally, uses a single colour font draw which when passed an x and y position in d0/d1 and a2 pointed at the text to display, is only capable of holding 5 digits at the present, but that should be easy to change.

Two: Add new levels to the game. Increase MAX_LEVELS to reflect the new number, and there you go.

Three: Change the font draw into a masked font draw, and give it the ability to display more than one colour.

Next month: enemies and music!

MASKS

At present the Ankhs are not being masked as they are drawn. This means that if they were placed in front of a platform, the platform would disappear. The most difficult area of programming animations to understand is the principle of masks. If you position a sprite on the screen, the block containing the sprite will affect the colours already there. To overlay the sprite properly against the background, you first need to create and position a mask which changes the screen colours in the area of the sprite block.

Now when you lay the sprite onto the screen, the colours of the mask change back again to the correct orginal colour.

Confused? A mask is a block of data which has a bit set for every pixel that is blank on the sprite block. The mask can be 'AND.W'ed with the background data, before you 'OR.W' your sprite data to draw it on the background.

If you look inside draw.s at the routine _draw_collectables you will see that we have no mask. You can calculate this when you draw your sprite by replacing the four move.w d?,?(a0) with the following piece of code.

Please note that, as throughout this month's piece, some of the comment lines have been wrapped to the next line.

```
move.w  d1,d5    ;copy plane 0 into d5
or.w    d2,d5    ;or plane 1 with d5
or.w    d3,d5    ;or plane 2 with d5
or.w    d4,d5
;or plane 3 with d5
not.w   d5
;invert d5 to create mask
and.w   d5,(a0)
;mask background (plane 0)
or.w    d1,(a0)
;combine with sprite plane 0
and.w   d5,PLANE_SIZE(a0)
;mask background (plane 0)
or.w    d2,PLANE_SIZE(a0)
;combine with sprite plane 0
and.w   d5,PLANE_SIDE*2(a0)
;mask background (plane 0)
or.w    d3,PLANE_SIZE*2(a0)
;combine with sprite plane 0
and.w   d5,PLANE_SIDE*3(a0)
;mask background (plane 0)
or.w    d4,PLANE_SIZE*3(a0)
;combine with sprite plane 0
```

Though, as you can see, this takes quite a lot of instructions, it would be much easier if we stored a copy of the mask with our data, then the top half of the above code would not be needed.

pg 154